

Strategies in P ρ Log

Besik Dundua

RISC, JKU Linz, Austria

bdundua@risc.uni-linz.ac.at

Temur Kutsia

RISC, JKU Linz, Austria

kutsia@risc.uni-linz.ac.at

Mircea Marin

University of Tsukuba, Japan

mmarin@cs.tsukuba.ac.jp

P ρ Log is an experimental extension of logic programming with strategic conditional transformation rules, combining Prolog with ρ Log calculus. The rules perform nondeterministic transformations on hedges. Queries may have several results that can be explored on backtracking. Strategies provide a control on rule applications in a declarative way. With strategy combinators, the user can construct more complex strategies from simpler ones. Matching with four different kinds of variables provides a flexible mechanism of selecting (sub)terms during execution. We give an overview on programming with strategies in P ρ Log and demonstrate how rewriting strategies can be expressed.

1 Introduction

P ρ Log (pronounced Pē-rō-log) is an experimental tool that extends logic programming with strategic conditional transformation rules, combining Prolog with ρ Log calculus [14]. ρ Log deals with hedges (sequences of terms), transforming them by conditional rules. Transformations are nondeterministic and may yield several results. Logic programming seems to be a suitable framework for such nondeterministic computations. Strategies provide a control on rule applications in a declarative way. Strategy combinators help the user to construct more complex strategies from simpler ones. Rules apply matching to the whole input hedge (or, if it is a single term, apply at the top position). Four different types of variables give the user flexible control on selecting subhedges in hedges (via individual and sequence variables) or subterms/contexts in terms (via function and context variables). As a result, the obtained code is usually quite short, declaratively clear, and reusable.

P ρ Log programs consist of clauses. The clauses either define user-constructed strategies by (conditional) transformation rules or are ordinary Prolog clauses. Prolog code can be used freely within P ρ Log programs. One can include its predicates in P ρ Log rules, which is especially convenient when arithmetic calculations or input-output features are needed.

P ρ Log inference mechanism is essentially the same as SLDNF-resolution, multiple results are generated via backtracking, its semantics is compatible with semantics of normal logic programs [13] and, hence, Prolog was a natural choice to base P ρ Log on: The inference mechanism comes for free, as well as the built-in arithmetic and many other useful features of the Prolog language. Following Prolog, P ρ Log is also untyped, but values of sequence and context variables can be constrained by regular hedge or tree languages. We do not elaborate on this feature here.

For the users familiar with logic programming and Prolog it is pretty easy to get acquainted with P ρ Log and to quickly start writing programs, since its syntax is very similar to that of Prolog and semantics is based on logic programming.

We tried to provide as little as possible hard-wired features in the system to give the user a freedom in experimenting with different choices. Probably the most notable such feature is the leftmost-outermost term traversal strategy the P ρ Log's matching algorithm uses, but it can also be easily modified since the corresponding Prolog code is open: Exchanging the order of clauses there would suffice. The user can also program different traversal strategies pretty easily inside P ρ Log.

The goal of this paper is to give an overview of $P\rho$ Log and, in particular, show how it uses strategies. After briefly reviewing the related work in Section 2, we discuss the syntax of $P\rho$ Log (Section 3) and then list some of the strategies from the library together with examples that explain the input-output behavior of the system (Section 4). Next, we show how user-defined strategies can be introduced illustrating it on the examples of defining rewriting strategies in $P\rho$ Log (Section 5). One can see that the code there is quite short and readable, and it also demonstrates expressiveness of $P\rho$ Log.

2 Related Work

Programming with rules has been experiencing a period of growing interest since the nineties when rewriting logic [15] and rewriting calculus [4, 5] have been developed and several systems and languages (ASF-SDF [2], CHR [8], Claire [3], ELAN [1], Maude [6], Stratego [18], just to name a few) emerged. The ρ Log calculus has been influenced by the ρ -calculus [4, 5] as also its name suggests, but there are some significant differences: ρ Log adopts logic programming semantics (clauses are first class concepts, rules/strategies are expressed as clauses), uses top-position matching, and employs four different kinds of variables. Consequently, $P\rho$ Log (based on ρ Log) differs from ELAN (based on ρ -calculus). Also, ELAN is a mature system with a very efficient compiler while $P\rho$ Log is an experimental extension of Prolog implemented in Prolog itself.

From the architecture point of view, $P\rho$ Log is closer to another mature system, CHR, because both extend the host language (in this case, Prolog) in a declarative way. However, the purpose of CHR is different: It has been designed as a language for writing constraint solvers. CHR extends Prolog with the rules to handle constraints that are the first class concept there. $P\rho$ Log is not designed specifically for programming constraint manipulation rules and we have not experimented with specifying such rules.

In the OBJ family of languages (OBJ2 [9], OBJ3 [11], CafeOBJ [10]), local strategies can be explicitly specified. They guide evaluation: In function calls only the arguments, specified by the strategies are evaluated. Among the other systems, strategic programming is supported Maude, and Stratego. Maude is based on rewriting logic, can perform efficient pattern matching modulo equational theories like associativity, commutativity, idempotence, and has a high-performance rewrite engine. Stratego is a domain-specific language designed for program transformation, combining term traversal strategies, pattern matching, and rewriting.

To compare with these systems, $P\rho$ Log has been designed with the purpose to experiment with strategic conditional transformation rules in a logic programming environment. Strategies and nondeterministic computations fit well into the logic programming paradigm. Bringing hedge and context pattern matching and strategic transformations into logic programs seems to facilitate writing declaratively clear, short, and reusable code.

3 Preliminaries

$P\rho$ Log is essentially based on the language of ρ Log [14], extending Prolog with it. Here we use the $P\rho$ Log notation for this language, writing its constructs in typewriter font. The expressions are built over the set of functions symbols \mathcal{F} and the sets of individual, sequence, function, and context variables. These sets are disjoint. $P\rho$ Log uses the following conventions for the variables names: Individual variables start with `i_` (like, e.g., `i_Var` for a named variable or `i_` for the anonymous variable), sequence variables start with `s_`, function variables start with `f_`, and context variables start with `c_`. The symbols

in \mathcal{F} , except the special constant `hole`, have flexible arity. To denote the symbols in \mathcal{F} , PpLog basically follows the Prolog conventions for naming functors, operators, and numbers.

Terms t and hedges h are constructed in a standard way by the following grammars:

$$\begin{aligned} t &::= i_X \mid \text{hole} \mid f(h) \mid f_X(h) \mid c_X(t) \\ h &::= t \mid s_X \mid \text{eps} \mid h_1, h_2 \end{aligned}$$

where `eps` stands for the empty hedge and is omitted whenever it appears as a subhedge of another hedge. $a(\text{eps})$ and $f_X(\text{eps})$ are often abbreviated as a and f_X . A *Context* is a term with a single occurrence of `hole`. A context can be applied to a term, replacing the hole by that term. For instance, applying the context $f(\text{hole}, b)$ to $g(a)$ gives $f(g(a), b)$.

A *Substitution* is a mapping that maps individual variables to (hole-free) terms, sequence variables to (hole-free) hedges, function variables to function symbols, and context variables to contexts so that

- all but finitely many individual, sequence, and function variables are mapped to themselves, and
- all but finitely many context variables are mapped to themselves applied to the hole.

The mapping is extended to arbitrary terms and hedges in the usual way. For instance, the image of the hedge $(c_Ctx(i_Term), f_Func(s_Terms1, a, s_Terms2))$ under the substitution $\{c_Ctx \mapsto f(\text{hole}), i_Term \mapsto g(s_X), f_Func \mapsto g, s_Terms1 \mapsto \text{eps}, s_Terms2 \mapsto (b, c)\}$ is the hedge $(f(g(s_X)), g(a, b, c))$.

In [12], an algorithm to solve *matching equations* in the language just described has been introduced. Matching equations are equations between two hedges, one of which does not contain variables. Such matching equations may have zero, one, or more (finitely many) solutions, called matching substitutions or *matchers*.

Example 1 The term $c_X(f(s_Y))$ matches $g(f(a, b), h(f(a), f))$ in three different ways with the matchers $\{c_X \mapsto g(\text{hole}, h(f(a), f)), s_Y \mapsto (a, b)\}$, $\{c_X \mapsto g(f(a, b), h(\text{hole}, f)), s_Y \mapsto a\}$, and $\{c_X \mapsto g(f(a, b), h(f(a), \text{hole})), s_Y \mapsto \text{eps}\}$.

The hedge $(s_X, f_F(i_X, a, s_), s_Y)$ matches $(a, f(b), g(a, b), h(b, a))$ with the matcher $\{s_X \mapsto (a, f(b), g(a, b)), f_F \mapsto h, i_X \mapsto b, s_Y \mapsto \text{eps}\}$.

A *pLog atom* (ρ -atom) is a triple consisting of a term st (a *strategy*) and two hedges $h1$ and $h2$, written as $st :: h1 ==> h2$. (The hedges $h1$ and $h2$ do not contain the `hole` constant.) Intuitively, it means that the strategy st transforms the hedge $h1$ to the hedge $h2$. (We will use this, somehow sloppy, but intuitively clear wording in this paper.) Its negation is written as $st :: h1 =\Rightarrow h2$. A *pLog literal* (ρ -literal) is a ρ -atom or its negation. A *PpLog clause* is either a Prolog clause, or a clause of the form $st :: h1 ==> h2 :- \text{body}$ (in the sequel called a ρ -clause) where body is a (possibly empty) conjunction of ρ - and Prolog literals.¹

A *PpLog program* is a sequence of PpLog clauses and a *query* is a conjunction of ρ - and Prolog literals. A restriction on variable occurrence is imposed on clauses: If a ρ -clause has the body that contains Prolog literals, then the only variables that can occur in those Prolog literals are the ρ Log individual variables. (When it comes to evaluating such Prolog literals, the individual variables there are converted into Prolog variables.) The same restriction applies to ρ -queries where Prolog literals occur. On the other hand, Prolog clauses can not contain any ρ Log variables. In short: ρ -clauses and queries can contain only ρ Log variables. Prolog clauses and queries can contain only Prolog variables.

¹In fact, PpLog clauses may have a more complex structure, when (some of) the literals are equipped with membership constraints, constraining possible values of sequence and context variables. Such constraints are taken into account in the matching process. For simplicity, we do not consider them in this paper.

Both a program clause and a query should satisfy a syntactic restriction, called well-modedness, to guarantee that each execution step is performed using matching (which is finitary in our language) and not unification (whose decidability is not known. It subsumes context unification whose decidability is a long-standing open problem [16].). To explain the essence of the problem, consider a query $\text{str} :: (\text{i_X}, \text{i_X}) ==> \text{i_X}$. It contains (two copies of) a variable in the left-hand side, which might give rise to an arbitrarily complex context unification problem $t_1 =^? t_2$, if there is a clause with the head of the form $\text{str} :: (t_1, t_2) ==> h$ and t_1 and t_2 are terms containing context variables. It can be that the unification problem has infinitely many unifiers (this might be the case also with sequence variables), which leads to computing infinitely many answers. Even worse, since the decision algorithm for context unification is not known, an attempt to compute context unifiers might run forever without detecting that there are no more unifiers.

All these cases are extremely undesirable from the computational point of view. Therefore, we would like to restrict ourselves to the fragment that guarantees a terminating finitary solving procedure. Matching is one of such possible fragments. Well-moded clauses and queries forbid uninstantiated variables to appear in one of the sides of unification problems and, hence, allow only matching problems. Queries like $\text{str} :: (\text{i_X}, \text{i_X}) ==> \text{i_X}$ above and clauses that might lead to such kind of queries are not allowed in PpLog.

More specifically, well-modedness for PpLog programs extends the same notion for logic programs, introduced in [7]: A mode for the relation $\cdot :: \cdot ==> \cdot$ is a function that defines the input and output positions of the relation respectively as $\text{in}(\cdot :: \cdot ==> \cdot) = \{1, 2\}$ and $\text{out}(\cdot :: \cdot ==> \cdot) = \{3\}$. A mode is defined (uniquely) for a Prolog relation as well. A clause is moded if all its predicate symbols are moded. We assume that all ρ -clauses are moded. As for the Prolog clauses, we require modedness only for those ones that define a predicate that occurs in the body of some ρ -clause. If a Prolog literal occurs in a query in conjunction with a ρ -clause, then its relation and the clauses that define this relation are also assumed to be moded.

Before defining well-modedness, we introduce the notation $\text{vars}(E)$ for a set of variables occurring in an expression E , and define $\text{vars}(E, \{p_1, \dots, p_n\}) = \bigcup_{i=1}^n \text{vars}(E|_{p_i})$, where $E|_{p_i}$ is the standard notation for a subexpression of E at position p_i . The symbol \mathcal{V}_{an} stands for the set of anonymous variables. A ground expression contains no variables. Then well-modedness of queries and clauses are defined as follows:

Definition 1 A query L_1, \dots, L_n is well-moded iff it satisfies the following conditions for each $1 \leq i \leq n$:

- $\text{vars}(L_i, \text{in}(L_i)) \subseteq \bigcup_{j=1}^{i-1} \text{vars}(L_j, \text{out}(L_j)) \setminus \mathcal{V}_{\text{an}}$.
- If L_i is a negative literal, then $\text{vars}(L_i, \text{out}(L_i)) \subseteq \bigcup_{j=1}^{i-1} \text{vars}(L_j, \text{out}(L_j)) \cup \mathcal{V}_{\text{an}}$.
- If L_i is a ρ Log literal, then its strategy term is ground.

A clause $L_0 : -L_1, \dots, L_n$ is well-moded, iff the following conditions are satisfied for each $1 \leq i \leq n$:

- $\text{vars}(L_i, \text{in}(L_i)) \cup \text{vars}(L_0, \text{out}(L_0)) \subseteq \bigcup_{j=0}^{i-1} \text{vars}(L_j, \text{out}(L_j)) \setminus \mathcal{V}_{\text{an}}$.
- If L_i is a negative literal, then $\text{vars}(L_i, \text{out}(L_i)) \subseteq \bigcup_{j=1}^{i-1} \text{vars}(L_j, \text{out}(L_j)) \cup \mathcal{V}_{\text{an}} \cup \text{vars}(L_0, \text{in}(L_0))$.
- If L_0 and L_i are ρ Log literals with the strategy terms st_0 and st_i , respectively, then $\text{vars}(\text{st}_i) \subseteq \text{vars}(\text{st}_0)$.

PpLog allows only well-moded program clauses and queries. There is no restriction on the Prolog clauses if the predicate they define is not used in a ρ -clause.

Example 2 The query $\text{str1} :: a \Rightarrow i_X, \text{str2} :: i_Y \Rightarrow i_Z$ is not well-moded, because the variable i_Y in the input position of the second subgoal does not occur in the output position of the first subgoal. On the other hand, $\text{str1} :: a \Rightarrow i_X, \text{str2} :: i_X \Rightarrow i_Z$ is well-moded.

If we change the last goal into $\text{str1} :: a \Rightarrow i_X, \text{str2} :: i_X \Rightarrow i_Z$, well-modedness will get violated again, because the variable i_Z , occurring in the negative literal, does not appear in the output position of the previous subgoal. Examples of well-moded queries involving negative literals are, e.g., $\text{str1} :: a \Rightarrow (i_X, i_Z), \text{str2} :: i_X \Rightarrow i_Z$ and $\text{str1} :: a \Rightarrow i_X, \text{str2} :: i_X \Rightarrow i_Z$.

For well-moded programs and queries, PpLog uses Prolog's depth-first inference mechanism with the leftmost literal selection in the goal. If the selected literal is a Prolog literal, then it is evaluated in the standard way. If it is a ρ -atom of the form $\text{st} :: h1 \Rightarrow h2$, then PpLog finds a (renamed copy of a) program clause $\text{st}' :: h1' \Rightarrow h2' :- \text{body}$ such that st' matches st and $h1'$ matches $h1$ with a substitution σ . Then, it replaces the selected literal in the query with the conjunction of $\text{body}\sigma$ and a literal that forces matching $h2$ to $h2'\sigma$, applies σ to the rest of the query, and continues. Success and failure are defined in the standard way. Backtracking allows to explore other alternatives that may come from matching the selected query literal to the head of the same program clause in a different way, or to the head of another program clause.

Negative ρ -literals are processed by the standard negation-as-failure rule: A negative query of the form $\text{str} :: h1 \Rightarrow h2$ succeeds if all attempts of satisfying its complementary literal, a positive query $\text{str} :: h1 \Rightarrow h2$, end with failure. Well-modedness guarantees that whenever such a negative literal is selected during the PpLog execution process, there are no variables in it except, maybe, some anonymous variables that may occur in $h2$.

4 Strategic Programming in PpLog

Strategies can be combined to express in a compact way many tedious small step transformations. These combinations give more control on transformations. PpLog provides a library of several predefined strategy combinators. Most of them are standard. The user can write her own strategies in PpLog or extend the Prolog code of the library. Some of the predefined strategies and their intuitive meanings are the following:

- $\text{id} :: h1 \Rightarrow h2$ succeeds if the hedges $h1$ and $h2$ are identical (or can be made identical by $h2$ matching $h1$) and fails otherwise.
- $\text{compose}(\text{st}_1, \text{st}_2, \dots, \text{st}_n), n \geq 2$, first transforms the input hedge by st_1 and then transforms the result by $\text{compose}(\text{st}_2, \dots, \text{st}_n)$ (or by st_2 , if $n = 2$). Via backtracking, all possible results can be obtained. The strategy fails if either st_1 or $\text{compose}(\text{st}_2, \dots, \text{st}_n)$ fails.
- $\text{choice}(\text{st}_1, \dots, \text{st}_n), n \geq 1$, returns a result of a successful application of some strategy st_i to the input hedge. It fails if all st_i 's fail. By backtracking it can return all outputs of the applications of each of the strategies $\text{st}_1, \dots, \text{st}_n$.
- $\text{first_one}(\text{st}_1, \dots, \text{st}_n), n \geq 1$, selects the first st_i that does not fail on the input hedge and returns only one result of its application. first_one fails if all st_i 's fail. Its variation, first_all , returns via backtracking all the results of the application to the input hedge of the first strategy st_i that does not fail.

- $\text{nf}(\text{st})$, when terminates, computes a normal form of the input hedge with respect to st . It never fails because if an application of st to a hedge fails, then $\text{nf}(\text{st})$ returns that hedge itself. Backtracking returns all normal forms.
- $\text{iterate}(\text{st}, N)$ starts transforming the input hedge with st and returns a result (via backtracking all the results) obtained after N iterations for a given natural number N .
- $\text{map1}(\text{st})$ maps the strategy st to each term in the input hedge and returns the result hedge. Backtracking generates all possible output hedges. st should operate on a single term and not on an arbitrary hedge. $\text{map1}(\text{st})$ fails if st fails on at least one term from the input hedge. map is a variation of map1 where the single-term restriction is removed. It should be used with care because of high nondeterminism. Both map1 and map , when applied to the empty hedge, return the empty hedge.
- interactive takes a strategy from the user, transforms the input hedge by it and waits for further user instruction (either to apply another strategy to the result hedge or to finish).
- $\text{rewrite}(\text{st})$ applies to a single term (not to an arbitrary hedge) and rewrites it by st (which also applies to a single term). Via backtracking, it is possible to obtain all the rewrites. The input term is traversed in the leftmost-outermost manner. Note that $\text{rewrite}(\text{st})$ can be easily implemented inside PpLog :

```
rewrite(i_Str) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
  i_Str :: i_Redex ==> i_Contractum.
```

We give below few examples that demonstrate the use some of the PpLog features, including the strategies we just mentioned. The users can define own strategies in a program either by writing clauses for them or using abbreviations of the form $\text{str}_1 := \text{str}_2$. Such an abbreviation stands for the clause $\text{str}_1 :: s_X ==> s_Y :- \text{str}_2 :: s_X ==> s_Y$.

Example 3 *Let str1 and str2 be two strategies defined as follows:*

```
str1 :: (s_1, a, s_2) ==> (s_1, f(a), s_2).
str2 :: (s_1, i_x, s_2, i_x, s_3) ==> (s_1, i_x, s_2, s_3).
```

Putting different strategies in the goal we get different answers:

- The goal $\text{str1} :: (a, b, a, f(a)) ==> s_X$ returns two answers (instantiations of the sequence variable s_X): $(f(a), b, a, f(a))$ and $(a, b, f(a), f(a))$. Multiple answers are computed by backtracking. They are two because (s_1, a, s_2) matches $(a, b, a, f(a))$ in two ways, with the matchers $\{s_1 \mapsto \text{eps}, s_2 \mapsto (b, a, f(a))\}$ and $\{s_1 \mapsto (a, b), s_2 \mapsto f(a)\}$, respectively.
- If we change the previous goal into $\text{str1} :: (a, b, a, f(a)) ==> (s_X, f(a), s_Y)$, then PpLog will return four answers that correspond to the following instantiations of s_X and s_Y :
 1. $s_X \mapsto \text{eps}, s_Y \mapsto (b, a, f(a))$.
 2. $s_X \mapsto (f(a), b, a), s_Y \mapsto \text{eps}$.
 3. $s_X \mapsto (a, b), s_Y \mapsto f(a)$.
 4. $s_X \mapsto (a, b, f(a)), s_Y \mapsto \text{eps}$.
- The goal $\text{str1} :: (a, b, a, f(a)) =\Rightarrow s_ fails, because it's positive counterpart succeeds. On the other hand, $\text{str1} :: (a, b, a, f(a)) =\Rightarrow (b, s_ succeeds.$$

- The composition `compose(str1, str2) :: (a, b, a, f(a)) ==> s_X` gives two answers: `(f(a), b, a)` and `(a, b, f(a))`,
- On the goal `choice(str1, str2) :: (a, b, a, f(a)) ==> s_X` we get three hedges as answers: `(f(a), b, a, f(a))`, `(a, b, f(a), f(a))`, and `(a, b, f(a))`.
- `nf(compose(str1, str2)) :: (a, b, a, f(a)) ==> s_X`, which computes a normal form of the composition, returns `(f(a), b)` twice, computing it in two different ways.
- The goal `first_one(str1, str2) :: (a, b, a, f(a)) ==> s_X` returns only one answer `(f(a), b, a, f(a))`. This is the first output computed by the first applicable strategy, `str1`.
- Finally, `first_all(str1, str2) :: (a, b, a, f(a)) ==> s_X` computes two instantiations: `(f(a), b, a, f(a))` and `(a, b, f(a), f(a))`. These are all the answers returned by the first applicable strategy, `str1`.

Example 4 The two PpLog clauses below flatten nested occurrences of the head function symbol of a term. The code is written using function and sequence variables, which makes it reusable, since it can be used to flatten terms with different heads and different numbers of arguments:

```
flatten_one :: f_Head(s_1, f_Head(s_2), s_3) ==> f_Head(s_1, s_2, s_3).
flatten := nf(flatten_one).
```

The first clause flattens one occurrence of the nested head. The second one (written in the abbreviated form) defines the `flatten` strategy as the normal form of `flatten_one`. Here are some examples of queries involving these strategies:

- `flatten_one :: f(a, f(b, f(c)), f(d)) ==> i_X` gives `f(a, b, f(c), f(d))`.
- `flatten :: f(a, f(b, f(c)), f(d)) ==> i_X` returns `f(a, b, c, d)`.
- We can map the strategy `flatten` to a hedge, which results in flattening each element of the hedge. For instance, the goal `map1(flatten) :: (a, f(f(a)), g(a, g(b))) ==> s_X` returns the hedge `(a, f(a), g(a, b))`.

Example 5 The `replace` strategy takes a term and a sequence of replacement rules, chooses a subterm in the given term that can be replaced by a rule, and returns the result of the replacement. `replace_all` computes a normal form with respect to the given replacement rules.

```
replace :: (c_Context(i_X), s_1, i_X -> i_Y, s_2) ==>
         (c_Context(i_Y), s_1, i_X -> i_Y, s_2).

replace_all :: (i_Term, s_Rules) ==> i_Instance :-
         nf(replace) :: (i_Term, s_Rules) ==> (i_Instance, s_).
```

With `replace_all`, one can, for example, compute an instance of a term under an idempotent substitution: `replace_all :: (f(x, g(x, y)), x -> z, y -> a) ==> i_X` gives `f(z, g(z, a))`. (We can take the conjunction of this goal with the cut predicate to avoid recomputing the same instance several times.) The same code can be used to compute a normal form of a term under a ground rewrite system, the sort of a term if the rules are sorting rules, etc.

Example 6 This is a bit longer example that shows how one can specify a simple propositional proving procedure in PpLog. We assume that the propositional formulas are built over negation (denoted by ‘-’) and disjunction (denoted by ‘v’). The corresponding PpLog program starts with the Prolog operator declaration that declares disjunction an infix operator:

```
:- op(200, xfy, v).
```

Next, we describe inference rules of a Gentzen-like sequent calculus for propositional logic. The rules operate on sequents, represented as `sequent(ant(sequence of formulas), cons(sequence of formulas))`. `ant` and `cons` are tags for the antecedent and consequent, respectively. There are five inference rules in the calculus: The axiom rule, negation left, negation right, disjunction left, and disjunction right.

```
axiom :: sequent(ant(s_, i_Formula, s_), cons(s_, i_Formula, s_)) ==> eps.
```

```
neg_left :: sequent(ant(s_F1, -(i_Formula), s_F2), cons(s_F3)) ==>
  sequent(ant(s_F1, s_F2), cons(i_Formula, s_F3)).
```

```
neg_right :: sequent(ant(s_F1), cons(s_F2, -(i_Formula), s_F3)) ==>
  sequent(ant(s_F1, i_Formula), cons(s_F2, s_F3)).
```

```
disj_left :: sequent(ant(s_F1, i_Formula1 v i_Formula2, s_F2), i_Cons) ==>
  (sequent(ant(s_F1, i_Formula1, s_F2), i_Cons),
   sequent(ant(s_F1, i_Formula2, s_F2), i_Cons)).
```

```
disj_right :: sequent(i_ant, cons(s_F1, i_Formula1 v i_Formula2, s_F2)) ==>
  sequent(i_ant, cons(s_F1, i_Formula1, i_Formula2, s_F2)).
```

Next, we need to impose control on the applications of the inference rules and define success and failure of the procedure. The control is pretty straightforward: To perform an inference step on a given hedge of sequents, we select the first sequent and apply to it the first applicable inference rule, in the order specified in the arguments of the strategy `first_one` below. When there are no sequents left, the procedure ends with success. Otherwise, if no inference step can be made, we have failure.

```
success :: eps ==> true.
```

```
inference_step :: (sequent(i_Ant, i_Cons), s_Sequents) ==>
  (s_New_sequents, s_Sequents) :-
  first_one(axiom, neg_left, neg_right, disj_left, disj_right) ::
  sequent(i_Ant, i_Cons) ==> s_New_sequents.
```

```
failure :: (sequent(i_Ant, i_Cons), s_Sequents) ==> false.
```

Finally, we specify the proof procedure as repeatedly applying the first possible strategy between success, inference_step, and failure (in this order) until none of them is applicable:

```
prove := nf(first_one(success, inference_step, failure)).
```

Note that it does matter in which order we put the clauses for the inference rules or the control in the program. What matters, is the order they are combined (e.g. as it is done in the strategy `first_one`).

What we described here is just one way of implementing the given propositional proof procedure in PpLog. One could do it differently as well, for instance, by writing recursive clauses like it has been shown in [14]. However, we believe that the version above is more declarative and naturally corresponds to the way the procedure is described in textbooks.

Note that there can be several clauses for the same strategy in a PpLog program. In this case they behave as usual alternatives of each other (when a query with this strategy is being evaluated) and are tried in the order of their appearance in the program, top-down.

5 Implementing Rewriting Strategies

In this section we illustrate how rewriting strategies can be implemented in PpLog. It can be done in a pretty succinct and declarative way. The code for leftmost-outermost and outermost rewriting is shorter than the one for leftmost-innermost and innermost rewriting, because it takes an advantage of PpLog's built-in term traversal strategy.

Leftmost-Outermost and Outermost Rewriting. As mentioned above, the `rewrite` strategy traverses a term in leftmost-outermost order to rewrite subterms. For instance, if the strategy `strat` is defined by two rules

```
strat :: f(i_X) ==> g(i_X).
strat :: f(f(i_X)) ==> i_X.
```

then for the goal `rewrite(strat) :: h(f(f(a)), f(a)) ==> i_X` we get, via backtracking, four instantiations for `i_X`, in this order: `h(g(f(a)), f(a))`, `h(a, f(a))`, `h(f(g(a)), f(a))`, and `h(f(f(a)), g(a))`.

If we want to obtain *only one result*, then it is enough to add the cut predicate at the end of the goal: `rewrite(strat) :: h(f(f(a)), f(a)) ==> i_X, !` returns only `h(g(f(a)), f(a))`.

To get *all the results of leftmost-outermost rewriting*, we have to find the first redex and rewrite it in all possible ways (via backtracking), ignoring all the other redexes. This can be achieved by using an anonymous variable for checking reducibility, and then putting the cut predicate:

```
rewrite_left_out(i_Str) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
    i_Str :: i_Redex ==> i_,
    !,
    i_Str :: i_Redex ==> i_Contractum.
```

The goal `rewrite_left_out(strat) :: h(f(f(a)), f(a)) ==> i_X` gives two instantiations for `i_X`: `h(g(f(a)), f(a))` and `h(a, f(a))`.

To return *all the results of outermost rewriting* we find an outermost redex and rewrite it. Backtracking returns all the results for all outermost redexes.

```
rewrite_out(i_Str) :: i_X ==> i_Y :-
    i_Str :: i_X ==> i_,
    !,
    i_Str :: i_X ==> i_Y.

rewrite_out(i_Str) :: f_F(s_1, i_X, s_2) ==> f_F(s_1, i_Y, s_2) :-
    rewrite_out(i_Str) :: i_X ==> i_Y.
```

The goal `rewrite_out(strat) :: h(f(f(a)), f(a)) ==> i_X` gives three answers, in this order: `h(g(f(a)), f(a))`, `h(a, f(a))`, and `h(f(f(a)), g(a))`.

Leftmost-Innermost and Innermost Rewriting. Implementation of innermost strategy in *PpLog* is slightly more involved than the implementation of outermost rewriting. It is not surprising since the outermost strategy takes an advantage of the *PpLog* built-in term traversal strategy. For innermost rewriting, we could have modified the *PpLog* source by simply changing the order of two rules in the matching algorithm to give preference to the rule that descends deep in the term structure. It would change the term traversal strategy from leftmost-outermost to leftmost-innermost. Another way would be to build term traversal strategies into *PpLog* (like it is done in ELAN and Stratego, for instance) that would give the user more control on traversal strategies, giving her a possibility to specify the needed traversal inside a *PpLog* program.

However, here our aim is different: We would like to demonstrate that rewriting strategies can be implemented quite easily inside *PpLog*. For the outermost strategy it has already been shown. As for the innermost rewriting, if we want to obtain *only one result by leftmost-innermost strategy*, we first check whether any argument of the selected subterm rewrites. If not, we try to rewrite the subterm and if we succeed, we cut the alternatives. The way how matching is done guarantees that the leftmost possible redex is taken:

```
rewrite_left_in_one(i_Str) :: c_Ctx(f_F(s_Args)) ==> c_Ctx(i_Contractum) :-
    rewrites_at_least_one(i_Str) :: s_Args ==> i_,
    i_Str :: f_F(s_Args) ==> i_Contractum,
    !.

rewrites_at_least_one(i_Str) :: (s_, i_X, s_) ==> true :-
    rewrite(i_Str) :: i_X ==> i_,
    !.
```

To get *all results of leftmost-innermost rewriting*, we first check whether the selected subterm is an innermost redex. If yes, the other redexes are cut off and the selected one is rewritten in all possible ways:

```
rewrite_left_in(i_Str) :: c_Context(f_F(s_Args)) ==>
    c_Context(i_Contractum) :-
    rewrites_at_least_one(i_Str) :: s_Args ==> i_,
    i_Str :: f_F(s_Args) ==> i_,
    !,
    i_Str :: f_F(s_Args) ==> i_Contractum.
```

If *strat* is the strategy defined in the previous section, then we have only one answer for the goal `rewrite_left_in(strat) :: h(f(f(a)), f(a)) ==> i_X: the term h(f(g(a)), f(a))`. The same term is returned by `rewrite_left_in_one`.

Finally, `rewrite_in` computes *all results of innermost rewriting* via backtracking:

```
rewrite_in(i_Str) :: f_F(s_Args) ==> i_Y :-
    rewrites_at_least_one(i_Str) :: s_Args ==> i_,
    i_Str :: f_F(s_Args) ==> i_Y.

rewrite_in(i_Str) :: f_F(s_1, i_X, s_2) ==> f_F(s_1, i_Y, s_2) :-
    rewrite_in(i_Str) :: i_X ==> i_Y.
```

The goal `rewrite_in(strat) :: h(f(f(a)), f(a)) ==> i_X` returns two instantiations of *i_X*: `h(f(g(a)), f(a))` and `h(f(f(a)), g(a))`.

6 Concluding Remarks

PpLog extends Prolog with strategic conditional transformation rules that operate on hedges. The rules, written as clauses in PpLog programs, define strategies. Strategy combinators help the user to construct more complex strategies from simpler ones. PpLog queries may have several results. They can be explored by backtracking. Four different kinds of variables used in PpLog make the system expressive and flexible.

PpLog is based on Prolog's inference mechanism and allows Prolog clauses and predicates in its programs. The users familiar with logic programming and Prolog can very quickly start using PpLog since its syntax is similar to that of Prolog and semantics is based on logic programming.

We gave a brief overview on strategies in PpLog, explained some of them on examples, and showed how rewriting strategies can be compactly and declaratively implemented. PpLog is written in SWI-Prolog [17] and has been tested for versions 5.6.50 and later. It is available for downloading from <http://www.risc.uni-linz.ac.at/people/tkutsia/software.html>.

7 Acknowledgments

This research has been partially supported by the European Commission Framework 6 Programme for Integrated Infrastructures Initiatives under the project SCIENCE—Symbolic Computation Infrastructure for Europe (Contract No. 026133) and by JSPS Grant-in-Aid no. 20500025 for Scientific Research (C).

References

- [1] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau & M. Vittek (1996): *ELAN: A logical framework based on computational systems*. *Electronic Notes in Theoretical Computer Science* 4, pp. 35–50.
- [2] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser & J. Visser (2001): *The ASF + SDF Meta-environment: A Component-Based Language Development Environment*. In: R. Wilhelm, editor: *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*, LNCS 2027. Springer, pp. 365–370.
- [3] Y. Caseau, F.-X. Josset & F. Laburthe (2002): *Claire: combining sets, search and rules to better express algorithms*. *Theory and Practice of Logic Programming* 2(6), pp. 769–805.
- [4] H. Cirstea & C. Kirchner (2001): *The rewriting calculus - Part I*. *Logic Journal of the IGPL* 9(3), pp. 339–375.
- [5] H. Cirstea & C. Kirchner (2001): *The rewriting calculus - Part II*. *Logic Journal of the IGPL* 9(3), pp. 377–410.
- [6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & J. F. Quesada (2002): *Maude: specification and programming in rewriting logic*. *Theoretical Computer Science* 285(2), pp. 187–243.
- [7] P. Dembinski & J. Maluszynski (1985): *AND-parallelism with intelligent backtracking for annotated logic programs*. In: *Proceedings of the 2nd IEEE Symposium on Logic Programming*. IEEE Computer Society, pp. 29–38.
- [8] T. Frühwirth (1998): *Theory and Practice of Constraint Handling Rules*. *J. Logic Programming* 37(1–3), pp. 95–138.
- [9] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud & J. Meseguer (1985): *Principles of OBJ2*. In: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL'85)*. ACM Press, pp. 52–66.

- [10] K Futatsugi & A. T. Nakagawa (1997): *An Overview of CAFE Specification Environment - An Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications over Networks*. In: *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE Computer Society, pp. 170–182.
- [11] J. A. Goguen, T. Winkler, K. Futatsugi, J. Meseguer & J.-P. Jouannaud (2000): *Introducing OBJ*. In: J. A. Goguen & G. Malcolm, editors: *Software Engineering with OBJ - Algebraic Specification in Action*. Kluwer Academic Publishers, pp. 3–167.
- [12] T. Kutsia & M. Marin (2005): *Matching with Regular Constraints*. In: G. Sutcliffe & A. Voronkov, editors: *Logic in Programming, Artificial Intelligence and Reasoning. Proceedings of the 12th International Conference LPAR'05, LNAI 3835*. Springer, pp. 215–229.
- [13] J. Lloyd (1987): *Foundations of Logic Programming*. Springer-Verlag, 2nd edition.
- [14] M. Marin & T. Kutsia (2006): *Foundations of the Rule-Based System pLog. J. Applied Non-Classical Logics* 16(1-2), pp. 151–168.
- [15] N. Martí-Oliet & J. Meseguer (2002): *Rewriting Logic: Roadmap and Bibliography*. *Theoretical Computer Science* 285(2), pp. 121–154.
- [16] *RTA List of Open Problems. Problem #90. Are context unification and linear second order unification decidable?* <http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html>.
- [17] *SWI-Prolog*. <http://www.swi-prolog.org>.
- [18] E. Visser (2001): *Stratego: A Language for Program Transformation Based on Rewriting Strategies*. In: A. Middeldorp, editor: *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA'01), LNCS 256*. Springer, pp. 357–362.